

Extracting URLs from JavaScript via Program Analysis

Qi Wang, Jingyu Zhou, Yizhou Zhang, Jianjun Zhao
School of Software

Shanghai Jiao Tong University, Shanghai, China
{aywq, westward.zhang}@sjtu.edu.cn, {zhou-jy, jj-zhao}@cs.sjtu.edu.cn

Abstract—With the extensive use of client-side JavaScript in web applications, web contents are becoming more dynamic than ever before. This poses significant challenges for search engines, because more web URLs are now hidden inside JavaScript code and most web crawlers are script-agnostic, significantly reducing the coverage of search engines. We present a hybrid approach that combines static analysis with dynamic execution, overcoming the weakness of a purely static or dynamic approach that either lacks accuracy or suffers from huge execution cost. We apply program analysis techniques such as statement coverage and range analysis to improve the performance of URL mining. Our approach can handle modern web applications with DOM interactions, dynamic code generation and JavaScript libraries. Experiments on large sets of popular web pages show that our approach is effective and efficient in practice.

I. INTRODUCTION

JavaScript is the predominant scripting language used in web browsers, which allows programmers to create rich user interfaces and sophisticated functionality for web applications. With the rise of Web 2.0, interactive web applications increasingly rely on the AJAX technology based on JavaScript. In fact, 98 out of the 100 most viewed websites in the United States contain JavaScript programs [1]. Another recent study [2] found that all top 100 sites and 89% of top 10,000 sites use JavaScript.

Unfortunately, the ubiquity of JavaScript, along with its dynamic nature, is now posing significant challenges for search engines, because a large amount of web information, including URLs, is now buried in client-side JavaScript code. Major search engines only analyze the static parts of the HTML documents, and have difficulty in indexing scripting parts, e.g. Google [3]. As a result, many URLs embedded in JavaScript code are missed, which may significantly reduce the coverage and quality of a search engine.

Client-side JavaScript poses many novel challenges for program analysis. Fig. 1 illustrates such an example of real JavaScript code. First, note that variables `index` and `lang` interact with the HTML elements via the Document Object Model (DOM), which means that we must provide HTML DOM for effective analysis. Second, the use of `eval` generates dynamic code for execution, which is difficult for static analysis. Other challenges include the event-driven model of code execution, the use of sophisticated JavaScript libraries, and accesses to the browser API via Browser Object Model (BOM).

```
var index = document.frm1.lang.selectedIndex; 1
var lang = document.frm1.lang.options[index].value 2
var a1 = "win", a2 = "dow.", a3 = "loca", a4 = "tion.", 3
    a5 = "replace", a6 = "lang.asp?lang=" + lang 4
var i, str = ""; 5
for(i=1; i<=6; i++){ 6
    str += eval("a"+i); 7
} 8
eval(str); 9
```

Fig. 1: An example of redirecting a page to another URL which cannot be detected by traditional web crawlers. The JavaScript code dynamically constructs another piece of code like `window.location.replace = URL`, where URL takes values set by user from HTML elements. The `eval` on the last line executes the generated code and performs the redirection.

A possible approach is to simply employ a browser or a JavaScript engine to execute JavaScript programs in HTML documents, instrumenting the engine to output URLs encountered during execution. However, a purely dynamic approach like this does not scale well because it can incur prohibitive execution overhead, as shown in Section VI. Another drawback is that only one control-flow path is executed, possibly missing many URLs in other paths.

Hence, it is natural to enlist the help of static program analysis methods. To this end, our work begins with a text-based approach that uses regular expressions to match potential URLs in the JavaScript code. Due to the lack of grammatical and semantic information of a JavaScript program, this approach introduces a high false positive rate. This observation leads to an AST-based approach, which first parses a JavaScript program to obtain its abstract syntax tree (AST) and then traverses the AST to look for AST nodes that may contain URLs. However, most real JavaScript programs use dynamic code generation techniques and often access HTML DOM and browser APIs. Static analysis is not able to tackle these dynamic aspects of JavaScript.

Based on these insights, our approach finally evolves into a hybrid approach that combines static program analysis and dynamic program execution, leveraging the advantages of both. We use static analysis techniques to reason about control flows of JavaScript programs, and implement a special JavaScript interpreter to execute JavaScript code. The interpreter models not only the JavaScript language but also the HTML DOM objects and Browser Objects, and is fault tolerant in the presence of broken scripts. Other program analysis techniques such as statement coverage and range analysis are applied later

to further improve the performance of the hybrid approach.

We conducted experiments on a diverse set of popular real-world web pages, and made performance comparisons of the various approaches mentioned above. Experimental results show that the hybrid approach is able to produce the largest number of unique URLs among these approaches in a reasonable amount of time.

To summarize, the key contributions of this work are:

- We present a detailed survey on the typical forms of embedding URLs in client-side JavaScript programs.
- We design and implement several approaches for extracting URLs in client-side JavaScript. In particular, the proposed hybrid approach uses static analysis to reason program control flows, and exploits dynamic executions to tackle dynamic features of JavaScript and to find precise URL values. Our JavaScript interpreter models JavaScript language, core libraries, HTML DOM and Browser objects.
- Our evaluation on top sites and popular pages shows that 39.4% more URLs are hidden in JavaScript code and there is evidence that JavaScript is used to lazily load page contents, indicating JavaScript code may provide much information for search engines. Experimental results also demonstrate the effectiveness the proposed hybrid approach.

The remainder of this paper is organized as follows. Section II introduces the inclusion of JavaScript code in web pages. Section III categorizes static and dynamic URLs, illustrating how URLs are used in JavaScript code. Section IV describes the evolution of our approaches in detail, and Section V discusses some implementation issues. Evaluation on top sites and popular web pages are presented in Section VI. Section VII discusses related work. Finally, Section VIII concludes with future work.

II. BACKGROUND ON JAVASCRIPT

There are a number of ways of embedding JavaScript code in an HTML document and we categorize them into statically included JavaScript code and dynamically included JavaScript code.

A. Statically Included JavaScript Code

Statically included JavaScript codes are directly embedded in an HTML document with the following forms:

- Embedding via the script tag. A pair of script tags places JavaScript code within an HTML document. The script element may appear any number of times. Scripts can be defined within the contents of the script element or in an external file. If the src attribute of a script element is not set, the contents of between `<script>` and `</script>` are interpreted as the script. Otherwise, the contents of the element should be ignored, and the script should be retrieved via the value of src and be included in the HTML document. Here is an example to show the two forms.

```
<script type="text/javascript">
    document.write("<p>" + Date() + "</p>");
</script>

<script type="text/javascript" src="jQuery.js" >
</script>
```

- Embedding in the attribute of an HTML element. A piece of JavaScript code can be defined as an event attribute (such as onload and onclick) of an HTML element. The script is executed when the corresponding event is fired. Additionally, a script can be embedded with a JavaScript prefix in any non-event HTML attribute. The script is executed when the attribute is accessed. Below is an example illustrating these two forms.

```
<input type="button" id="regnow" value="Register"
    onclick="window.location='register.asp' />

<a href="javascript:refreshCaptcha_login();">login</a>
```

B. Dynamically Included JavaScript Code

This type of JavaScript codes is generated or included as the result of the execution of some JavaScript statements:

- eval-like functions. JavaScript allows dynamic construction of program code from text strings. The eval function is the primary way to carry out this task, which takes a string as the input parameter, interprets the string as JavaScript code and executes it. The string can contain an expression, a statement, or a sequence of statements. Similar to eval, the setTimeout and setInterval methods of the window object also take a string as JavaScript code and executes it. Additionally, the constructor of Function object also allows for dynamic construction of program code from string text. The following lists the syntax of these functions, where code is a string representing a piece of JavaScript that may only be determined at runtime.

```
eval(code);
window.setTimeout(code, millisec);
window.setInterval(code, millisec);
var func = new Function(arg1, arg2, ..., argN, code);
```

The following code contains actual examples.

```
function clock(){
    var t = new Date()
    document.getElementById("clock").value = t
}
eval("x=10;y=20;document.write(x*y)")
setTimeout("alert('5 seconds!')", 5000);
setInterval("clock()", 50)
var doAdd = new Function("iNum", "alert(iNum + 10);");
doAdd(10);
```

- Manipulating HTML DOM. Each HTML document loaded into a browser becomes a document object and each HTML element becomes an HTML DOM object. The write and writeln methods of the document object write the string representation of their arguments to an HTML document, where the value of the argument can be formatted HTML that contains statically included JavaScript code as we described above, thus effectively injecting JavaScript code into the HTML page. The

following is an example in which an HTML element containing JavaScript code is written to the document using the `writeln` method.

```
document.writeln("<a href='retrieveAccount.jsp' "+
"onclick='clearup(" + uid + ")'>Retrive Account</a>");
```

A script can also be included by adding a new script node or modifying the content of existing HTML elements. The following is an example of using a dynamically created script tag to collect visitor statistics:

```
var ga = document.createElement('script');
ga.type = 'text/javascript';
ga.async = true;
ga.src = ('https:' == document.location.protocol ?
'https://ssl' :
'http://www') + '.google-analytics.com/ga.js';
document.documentElement.firstChild.appendChild(ga);
```

III. A TAXONOMY OF URLS IN A WEB PAGE

We categorize URLs in a web page into two types: *static* and *dynamic* URLs. Static URLs are those directly embedded in HTML code in the form of tag properties, which can easily be extracted by a web crawler. Dynamic URLs occur in JavaScript code, where the exact location and value may only be known at execution time. In the following, we detail these two types of URLs.

A. Static URLs

Table I summarizes HTML elements (or tags) and corresponding attributes that can embed URLs in the W3C HTML 4.01 Specification [4]. For instance, the `href` attribute of an `<a>` tag specifies the URL of the linked page.

TABLE I: Attributes of HTML elements that are URLs.

HTML Element	Attribute	Description
form	action	server-side form handler
body	background	texture tile for document background
a, area, link	href	URL for linked resource
base	href	URL that acts as base URL
frame, iframe, img	longdesc	link to long description
script	src	URL for an external script
frame, iframe	src	source of frame content
img	src	URL of image to embed

Apart from those listed in Table I, the `meta` element can also embed URLs as attribute values. Examples are:

```
<meta http-equiv="refresh" content="5;URL=http://ask.com"/>
<meta http-equiv="location" content="URL=http://msn.com" />
```

B. Dynamic URLs

A meaningful URL can only occur in certain places in client-side JavaScript codes. We call these places as *URL-relevant points* which have the following forms:

- Method calls of the browser objects. Some methods of curtain browser objects deal with URLs, where one argument is intended to be a URL. The following is a summary of this type of methods:

```
window.open(url, name, features, replace)
window.navigate(url)
location.assign(url)
location.replace(url)
```

- Property assignments of browser objects. A browser's navigating behaviors can also be achieved by setting specific properties of browser objects, e.g., `document.location`. The following lists these object properties:

```
window.location = url
location.href = url
document.URL = url
document.location = url
```

- Property assignments of DOM objects. An HTML DOM object represents an element in an HTML document. Setting the property of a DOM object as specified in Table I means changing the URL for the property. Such a program statement is the point we are interested in. Property setting can be achieved by direct property assignment or by calling the `setAttribute` method of a DOM object. We show examples in the code below. An object name on the left of the property-set operator (dot) indicates the type of the corresponding HTML element.

```
anchorElement.href = url
frameElement.src = url
anchorElement.setAttribute("src", url)
```

- Sending of AJAX request. The `XMLHttpRequest` object may communicate with a web server in the background. For each AJAX request, the `XMLHttpRequest` object must specify a URL on the server:

```
xmlhttp.open(method, url, asyncthe )
```

- Dynamically injected HTML. Because JavaScript code can dynamically inject or modify HTML and HTML code often contains URLs, we must pay attention to such places:

```
document.write(html)
document.writeln(html)
htmlElement.innerHTML = html
```

In this paper we focus on finding the above dynamic URLs hidden in JavaScript code.

IV. APPROACHES

This section describes several approaches we have implemented to uncover dynamic URLs in details, which illustrates the evolution of our research for analyzing URLs in client-side JavaScript code.

A. The Text-based Approach

The client-side JavaScript code, though embedded in an HTML document in different forms, is plain text. Thus, a simple solution is to use some text-based patterns to search for dynamic URLs in the JavaScript code. This is feasible because many URLs existing in JavaScript are in the form of string literals [5].

We implemented a text-based approach that employs regular expressions to identify URL string literals in JavaScript code. Initially, we used the pattern provided by John Gruber [6] to match URLs. His regex pattern, as shown in Fig. 2, attempts to match any sort of URLs in an arbitrary string of text.

```
(?xi)
\b
# Capture 1: entire matched URL
(
  (?:
    # URL protocol and colon
    [a-z] [\w-]+:
    (?:
      # 1-3 slashes
      /{1,3}
      # or
      |
      # Single letter or digit or '%'
      [a-z0-9%]
      # (Trying not to match e.g. "URI::Escape")
    )
  )
  |
  # "www.", "www1.", "www2." "www999."
  www\d{0,3}[.]
  |
  # looks like domain name followed by a slash
  [a-z0-9.\-]+\.[a-z]{2,4}/
)
# One or more:
(?:
  # Run of non-space, non-()<>
  [^\s()<>]+
  |
  # balanced parens, up to 2 levels
  \(((^\s()<>+|(\([^\s()<>+\)))*\))
)+
)
# End with:
(?:
  # balanced parens, up to 2 levels
  \(((^\s()<>+|(\([^\s()<>+\)))*\))
  |
  # not a space or one of these punct chars
  [^\s'!() \[\]{};:'.<>?]
)
)
```

Fig. 2: John Gruber's regex pattern for matching URLs using the extended multiline regex format.

Though very practical and powerful¹, the pattern doesn't work well on text containing JavaScript code. In our final implementation of the text-based approach, we define four groups of regular expressions. Each group is used to match a certain type of URL-relevant strings:

- URL-relevant points as defined in Section III-B. For example:

```
window.location = "http://stap.sjtu.edu.cn";
window.open("/index.html");
```

The Java regular expressions for this type are:

```
//method calls
(window.open|navigate)| (location.(assign|replace))|
document.write(ln)? \\s*\\(\\s*(['"]*)?\\)|\\(\\s*["']*)?\\)

//property assignments
((window|document)\\. (location|URL))|\\(\\s*\\.+\\.
(href|src|innerHTML)) \\s*=\\s*(['"]*)?\\)|\\(\\s*["']*)?\\)

//AJAX request
```

```
[^\\.]\\.\\.open\\s*\\(\\s*(['"]*)?\\(get|post) ['"]*)\\s*,
\\s*(['"]*)?\\)|\\(\\s*["']*)?\\)
```

- String literals starting with "http://" or "https://". For example:

```
var index = "http://www.google.com";
var obj = {name:'yahoo', src:'https://www.yahoo.com'}
```

The Java regular expression for this type is:

```
\"(https?:/[^\"]*)\" | '(https?:/[^\']*?)'
```

- String literals ending with ".html", ".htm", ".jsp", etc. For example:

```
var hot_pages = ['index.htm', 'sports.jsp', '/news/USA.aspx'];
```

The Java regular expression for this type is:

```
\"[^\"]+?\\. (s?html?|[js|ph]p|aspx?)\"
|
'[^']*+?\\. (s?html?|[js|ph]p|aspx?)'
```

- URLs in string literals that are pieces of HTML code. For example:

```
D.create('')
```

The Java regular expression for this type is:

```
\"<[a-z]+[>]*? (href|src) \\s*?=\\s*(['\"]*)?
\\s*(['>]*)? ([^\"]|\\s>|\\s<)*>\"
|
'<[a-z]+[>]*? (href|src) \\s*?=\\s*(['\"]*)?
\\s*(['>]*)? ([^\"]|\\s>|\\s<)*>'
```

B. The AST-based Approach

The problem of the text-based approach is its accuracy. Take the following code as an example:

```
window.open("/news" + Math.floor(Math.random()*10) + ".html")
```

The text-based approach will identify either /news or /news"+Math.floor(Math.random()*10)+".html as a URL depending on the regular expressions it uses to match URLs for window.open. The former is recognized with quotes as string delimiters, and the latter uses parentheses. Both are apparently incorrect.

To correctly identify the URL in the example, we actually need syntactic information, i.e., "/news", Math.floor(Math.random()*10), and ".html" which are parts of the URL between the left and the right parentheses. To this end, we can exploit the abstract syntax tree (AST), which is a tree representation of the syntactic structure of a program, where inner nodes of the tree are correlated with statements or expressions and the leaf nodes of the tree serve as variables and constants. With the knowledge of AST, we can determine that the URL in the example is a concatenation of the values of those three parts.

We have implemented the following two AST-based approaches:

The AST Literal Approach. The AST literal approach aims at finding string literals representing URLs. This approach traverses an AST in a depth-first manner. The traversal the AST works as follows. If the current node is a string literal node, similar to the text-based approach, we determine if it

¹<http://daringfireball.net/misc/2010/07/url-matching-regex-test-data.text>

Easily Computable (e) ::= $l \mid unop\ e \mid e\ binop\ e \mid fc \mid p$
Literal (l) ::= $string \mid number \mid true \mid false$
 $unop \in \{+, -, !, \sim\}$
 $binop \in \{+, -, \times, /, \%, ==, !=, <, >, \dots\}$
Type (τ) ::= $string \mid number \mid boolean$
Function Call (fc) ::= $f(e_1, \dots, e_n)$ and $f \in \mathcal{F}$
 $\mathcal{F} = \{\text{functions of } Math \text{ and } \tau\} \cup \{\text{global functions}\}$
Property Get (p) ::= $\{\text{properties of } Math \text{ and } \tau\}$

Fig. 3: Definition of easily computable AST nodes.

represents a URL by pattern matching. If the node is a function call node or assignment node, we first check if this node contains URL-relevant points (Section III-B). If so and the AST node for the value of the URL-relevant point is a single string literal node, we can faithfully take the string as a URL. Otherwise, we stop further traversal on this node and continue with the next node, because the correct value of the URL needs to be computed with more context information. In this way, we avoid making errors as the text-based approach.

The AST Simple Computation Approach. The above approach misses a number of URLs when the URL-relevant point is not a string literal. To address this problem, instead of just checking whether a node is a single string literal node or not, we check if the node is *easily computable*. If so, we compute the value of the node and take the value as a URL.

Fig. 3 illustrates the formal definition of easily computable. An easily computable AST node is the one whose value can be computed without any context information, i.e., no dependence on variables. Specifically, such a node can be literals, expressions of literals, function calls, and property gets. Both unary and binary operations are supported. For function calls and property gets, JavaScript building objects (e.g., `Math` and `String`) are supported. We also implement a number of JavaScript global functions.

This approach can correctly compute the URL for the above example at the beginning of this subsection.

C. The Hybrid Approach

While being more accurate, the AST-based approach has difficulty in dealing with dynamic features offered by the JavaScript language such as dynamic code generation and DOM interactions. For instance, code in Fig. 1 cannot be analyzed by the AST-based approach.

To tackle the challenges posed by the dynamic features of JavaScript, our approach finally evolves into a hybrid approach, combining static program analysis and dynamic program execution together. Fig. 4 illustrates the steps of our hybrid approach. For a given web page, we first use static analysis to build an AST and a call graph (CG) for JavaScript code. A DOM tree is also constructed for the HTML document. Then, we use a lightweight JavaScript interpreter to simulate JavaScript executions, emitting URLs at URL-relevant points.

The interpreter is developed to faithfully model the JavaScript language and the core library as specified in the

ECMAScript standard. During execution, the interpreter performs different actions when visiting various AST nodes to approximate the dynamic behavior, based on the information of dynamic scoping and value typing. We do not model heap and as a result garbage collection is not a concern. Another benefit of manipulating AST rather than intermediate representation is that we can conveniently retrieve the value of a variable since variables keep their original names in AST.

Extracting and Assembling JavaScript. Client-side JavaScript code fragments exist in different forms and places in an HTML document. To perform analysis on client-side JavaScript code, we must first extract all the code fragments and assembling them into a complete and compilable JavaScript code.

For statically included JavaScript (Section II-A), we employ regular expressions to identify and extract scripts. Dynamically included JavaScript is handled by the JavaScript interpreter in later steps of our approach.

Sequence of Scripts. A single HTML document may contain more than one pair of non-overlapping script tags [7]. We assemble all JavaScript code fragments together and preserve their order in the original HTML document.

Common JavaScript Libraries. We don't include common JavaScript libraries like jQuery², Prototype³ and YUI⁴ in the resultant assembled JavaScript program. Instead, we only record the existence of these libraries. Currently, we identify the existence of these libraries by analyzing the URL and the content of the external JavaScript file. Common JavaScript libraries are treated in a different way as discussed in Section V-D.

Wrapping Scripts Defined in HTML Element Attributes. As discussed in Section II-A, JavaScript code can be embedded in the attribute of an HTML element. In fact, such a usage signifies a new entry point to the whole program, and the execution scope chain that is used to resolve variables includes all DOM objects on the path from the HTML element to the root of the document [8].

We wrap each piece of this kind of code into a separate function and record the path to the HTML element as well as the attribute the code resides in. In this way, scripts embedded in HTML elements can be easily recognized and receive special treatment in execution.

In summary, the algorithm for extracting and assembling JavaScript code is described in Algorithm 1. `Pattern1` through `Pattern4` refer to the forms for statically embedding JavaScript as discussed in Section II-A.

Generating AST and Constructing Call Graph (CG). We take assembled JavaScript code as input and parse it into an AST, and then construct call graph using the AST as input. Details of our implementations are discussed in Section V-B and Section V-C.

Modelling the HTML DOM and BOM. Client-side

²<http://jquery.com/>

³<http://www.prototypejs.org/>

⁴<http://yuilib.com/>

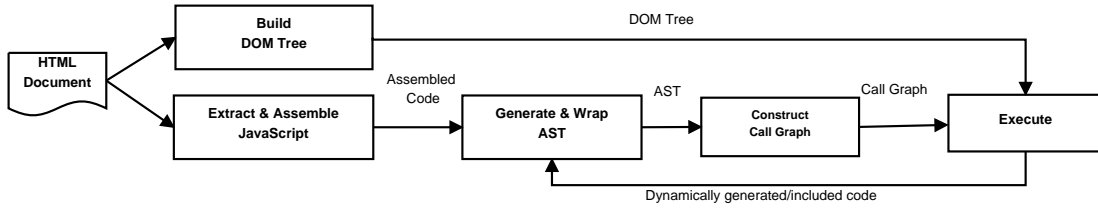


Fig. 4: Workflow of the hybrid approach.

Algorithm 1 Extract and assemble JavaScript code.

Input: the HTML document *html*

Output: the assembled script *code*, identified libraries *libs*

code = an empty string buffer

libs = \emptyset

for each script tag matches *Pattern1* or *Pattern2* **do**

if tag matches *Pattern1* **then**

 append scripts in the tag to *code*

else

src = the *src* attribute value of the tag

content = *RetriveExternalJS(src)*

 append *content* to *code*

end if

end for

for each attribute matches *Pattern3* or *Pattern4* **do**

 wrap code embedded in the attribute into a function

 append the function to *code*

end for

function RETRIVEEXTERNALJS(*src*)

lib = *IdentifyLibBySrc(src)*

if *lib* is not supported library **then**

content = retrieve the external file

lib = *IdentifyLibByConent(content)*

if *lib* is not supported library **then**

return *content*

end if

end if

libs = *libs* \cup *lib*

return empty string

end function

JavaScript uses Document Object Model (DOM) interfaces to manipulate and interact with the HTML document and uses Browser Object Model (BOM) interfaces to interact with the browser. URLs are often formed via the interaction of these interfaces. However, both the HTML DOM and BOM typically are not provided in JavaScript engines such as Chrome V8 [9], Rhino [10], and SpiderMonkey [11]. As a result, only using a JavaScript engine to analyze these URLs becomes impossible.

We have implemented the HTML DOM and BOM interfaces in our interpreter to model the host environments. We define a separate class for each type of HTML DOM element and model the DOM according to the behavior of

Firefox ⁵. Properties and methods of each DOM element are implemented. For the BOM objects, we only model the window, location and navigator objects.

Execution. JavaScript has multiple entry points and browsers execute JavaScript in an event-driven fashion. Some of the script runs as soon as the page completes loading, while other pieces run only in response to events. It is important to model the fact that *load handlers* are executed before the other kinds of *event handlers* [8]. Every event handler is an entry point of the program, but our analysis cannot predict the exact invocation sequence of event handlers, because it involves user interactions. Fortunately, the execution order of event handlers seldom affects the result of URL mining in our experience and is not crucial for the precision of a static analysis [8]. As a result, our analysis just randomly picks a sequence and executes it.

Algorithm 2 illustrates the execution sequence of our JavaScript interpreter, where *main* represents the top-level code which is the main entry of a program. Given an AST and a CG, the interpreter first executes the top-level code and load handlers. Then, the interpreter randomly chooses an event handler and executes it. When executing, the interpreter adds dynamically generated event handlers (i.e., via calls to *addEventListener* or assignments to event attributes) to EH, which contains unvisited event handlers. Finally, to improve coverage, the interpreter executes all functions in CG that has not been visited.

eval-like functions. The *eval* function is a popular way to dynamically inject JavaScript code — a study shows 59% of the most popular websites used *eval* [2]. During the execution, we capture the actual value that is passed to the *eval* function, and then parse the value to build an AST for the dynamically included JavaScript code. Finally, we insert the root of this syntax tree to the current point of the program AST and start traverse on this injected AST. Other *eval-like* functions such as *window.setTimeout()* and *window.setInterval()* are handled in the same manner.

If the dynamic code injects external JavaScript files during execution, the interpreter will download these files, parse them, and add the resulting AST back.

Loops and Recursions. Sometimes, execution may be trapped in an infinite loop or loops with many iterations,

⁵The DOM interfaces for manipulating web pages are not part of the ECMAScript standard. They are defined by a separate standardization effort by the W3C organization. In practice, browser implementations may differ from the standards and from each other in a few minor ways.

Algorithm 2 Pseudo-code of interpreter's execution.

```
Input: the AST ast, the call graph CG

VF = ∅           // visited functions
EH = {statically identified event handlers}

VisitFunction(main)
for each load handler l ∈ EH do
    VisitFunction(l)
    EH = EH - l
end for
while EH ≠ ∅ do
    randomly pick a handler l from EH
    VisitFunction(l)
    EH = EH - l
end while
for each f ∈ CG.getAllFunctions() \ VF do
    VisitFunction(f)
end for

function VISITFUNCTION(func)
    VF = VF ∪ {func}
    execute func
    EH = EH ∪ {dynamically identified event handlers}
end function
```

which may block the interpreter or significantly slow down the execution. Hence, a tradeoff between execution efficiency and accuracy is required. Our strategy is to define the maximum number of iterations of a loop. If the number of iterations exceeds this threshold, the loop is forced to quit. Similarly, we also set a threshold for the recursion depth of function calls. This may sometimes affect precision, but is an efficient method. Infinite loop or infinite recursion introduced by programming errors can be bypassed.

D. Further Improvement

To further improve the performance of our hybrid approach, we introduce two techniques, i.e., statement coverage and value range analysis.

Statement Coverage. A problem with purely dynamic execution by a normal browser engine or a JavaScript engine is that they are limited by the coverage. In other words, they can execute only one path of the program. As a result, statements in other control flow paths may not be executed, which misses URLs in these paths.

In order to improve the execution coverage, we implemented a *statement coverage* technique [12]. Specifically, at run-time, before the interpreter returns the results of executing a branch structure (e.g. *if-then-else*), the interpreter is forced to execute other unvisited branches in cloned context scopes. After all branches have been visited, the interpreter returns with the execution results stored to the upper AST node.

The code below shows an example that improving statement coverage is beneficial to mining more URLs. The *recommend* variable contains the URL string. Assuming today is Tuesday, the *else* branch of the *if* statement is taken. As a result, *recommend*'s value becomes string "weekdayRec.asp" and we find one URL. With statement coverage, another URL can be found on the other branch of the *if* statement.

```
var recommend;
```

```
var today = new Date();
var isWeekend = (today == 6 || today == 7);
if (isWeekend)
    recommend = "weekendSpecial.asp";
else
    recommend = "weekdayRec.asp";
document.write("<a href=\"" + recommend +
    "\">Today's Recommendation</a>");
```

When execution reaches the *if* AST node, the interpreter makes a snapshot of the current scope, which records variables declared and variables defined together with their values at that point of execution. After executing the *else* branch, the interpreter records the execution result and then forces to execute the *then* branch in the snapshotted scope. When statements in both branches are visited, the interpreter returns the execution result recorded to the upper AST node. With statements in both of the two branches covered by the interpreter, the *recommend* variable now has a set of string values: {"weekendSpecial.asp", "weekdayRec.asp"}.

In addition to *if-then-else*, other control flow structures including *switch-case*, *for* loop, *for-in* loop, *while* loop, *do-while* loop and *try-catch-finally*, as well as conditional expressions, are handled in similar ways.

Note that we do not intend to achieve high path coverage, which is too expensive to use. A relatively high statement coverage is sufficient enough for our URL mining task, as shown in Section VI.

Range Analysis. A simple range analysis can be helpful sometimes. Take the code below for an example. Because there is no definition to *hrs* from the condition of the *if* statement to the use of *hrs*, it can be inferred that the variable *hrs* can have integer values ranging from 8 to 23 at the program spot *images[hrs]*. Therefore, by applying range analysis, we can find 14 more URLs in this case.

```
var hrs = new Date().getHours();
if (hrs>=8 && hrs<=23)
    window.open("images/entry" + images[hrs]);
else
    window.open("images/entry" + defaultIndex);
```

Data types of number, boolean and string are candidates for range analysis, other data types like *object* are not considered.

V. IMPLEMENTATION

We have implemented all approaches in Java. This section discusses some implementation issues.

A. JavaScript Interpreter

We have decided to develop a lightweight JavaScript interpreter rather than instrumenting or modifying a JavaScript engine or a browser engine for several reasons. First, to support statement coverage and value range analysis, we designed the value of a variable in our interpreter to be a set (i.e., a variable is allowed to have multiple distinct values at a time), instead of strictly being a value. Operations on a variable are performed on each value in the variable's value set. Taking the following code as an example, we focus on the value of *couseCode*

which is a URL-relevant point. Supposing somehow from previous analysis we determine variable `course` can be string value "CS" or "SE" and variable `code` can be number value 101 or 201, then the value of `courseCode` is string concatenations between elements of the two sets, which will yield a new value set containing of string values: {"CS101", "CS201", "SE101", "SE201"}. However, such design would require substantial changes if it is applied on existing JavaScript or browser engines.

```
var course, code;
.....
.....
courseCode = course + code
location.href = courseCode;
```

Second, implementing a JavaScript interpreter allows us to manipulate the control flow in the execution to implement the statement coverage mechanism as described in Section IV-D.

Third, we intend to employ program slicing to remove JavaScript codes that are irrelevant to our analysis. A self-implemented interpreter gives us the flexibility to incorporate such technique. Moreover, supporting JavaScript libraries via matching rules (Section V-D) and being strong fault-tolerant also require such an interpreter.

Nevertheless, instead of building a JavaScript interpreter from scratch, our interpreter is implemented as a wrapper of Rhino. In our interpreter, JavaScript built-in objects and global functions are implemented by using their counterparts in Rhino.

Typing.

Fig. 5 shows the Java class inheritance hierarchy for JavaScript types. We define a separate class inheriting from the `JSValue` superclass for each runtime type in JavaScript. For example, `JSNumber` object is defined for numeric values, and `JSRegExp` is defined for JavaScript regular expression. JavaScript accepts higher-order functions; therefore, we define `JSFunction` for function objects and treat functions as first-class citizens. Besides basic operations on these value types, implicit conversions between different value types are implemented.

To support statement coverage and value range analysis, we define a `JSMultiValue` class that inherits `JSValue` and contains a set of `JSValue` objects.

Native/Built-in JavaScript Objects. JavaScript defines a set of native function objects, such as `Array`, `Math`, and `RegExp`. We treat each of them as a special `JSNative` object and provide implementations. Global properties like `NaN`, `undefined` and global functions such as `encodeURIComponent`, `parseInt` are also implemented.

Scoping and Prototype Chains. An `ExecuteScope` object is created upon each function call and the global entry as well. This scope object records the set of declared variables in the scope, as well as the set of variables defined in the scope and their values. Each scope object contains a pointer to its parent scope. When a variable is evaluated that does not refer

to a local variable, we walk up the scope chain to locate that variable. `this` object is also maintained in `ExecuteScope`.

JavaScript's inheritance mechanism is prototype-based, and we implement prototype chains of JavaScript objects.

Fault Tolerance. The real intention of the interpreter is not to execute JavaScript programs, but to help extract URLs. In a browser or JavaScript engines such as Rhino, a running JavaScript program may exit upon exceptions (e.g., using a variable that is not defined). However, our interpreter is designed to be fault-tolerant, and always continues to execute the rest of the program after encountering exceptions. Moreover, the interpreter does not consider some nitty-gritty features of JavaScript that are unlikely to appear in real JavaScript programs. When they do appear, the interpreter will simply ignore them and continue its execution.

B. Generating and Wrapping AST

We use the `Compiler` class (in package `com.google.javascript.jscomp`) of Google's Closure Compiler [13] to parse JavaScript codes into ASTs defined by Closure Compiler.

However, the nodes of the AST generated by Closure Compiler are all instances of the same `Node` class, which only uses a `type` field to distinguish different types of AST nodes. To provide easy manipulation of the AST, we define a wrapper class inheriting the `ASTNode` super class for each type of AST node and implement helpful methods. For example, we define a `getArguments` method for the `FunctionCall` node, which returns a list of AST nodes representing the arguments to the function call. Moreover, we define a `toSource` method for each type of AST node, which turns an AST node back to the corresponding formatted JavaScript source.

C. Constructing Call Graph

Again, we utilize Closure Compiler to construct call graphs. The `process` method of the `CallGraph` class (exists in package `com.google.javascript.jscomp`), which takes an AST generated by Closure Compiler as input, builds a call graph for the given AST. Such call graph connects functions to call sites and vice versa.

D. Supporting Common JavaScript Libraries

A JavaScript library often provides class-like abstractions and advanced GUI widgets and effects, and simplifies common tasks such as DOM manipulation and AJAX communication. The widespread use of JavaScript libraries in real-world websites warrants considerations in our analysis. However, JavaScript libraries are often large in size and use unusual coding tricks which make the analysis difficult and imprecise.

Our approach deals with common JavaScript libraries in a different way. Rather than analyzing and executing the code of a common JavaScript library, we identify URL-relevant points in the library APIs and define matching rules for them. If a certain kind of library is identified, the matching rules for the library will be checked when a function call or an assignment occurs. This strategy reduces more than 70% of JavaScript

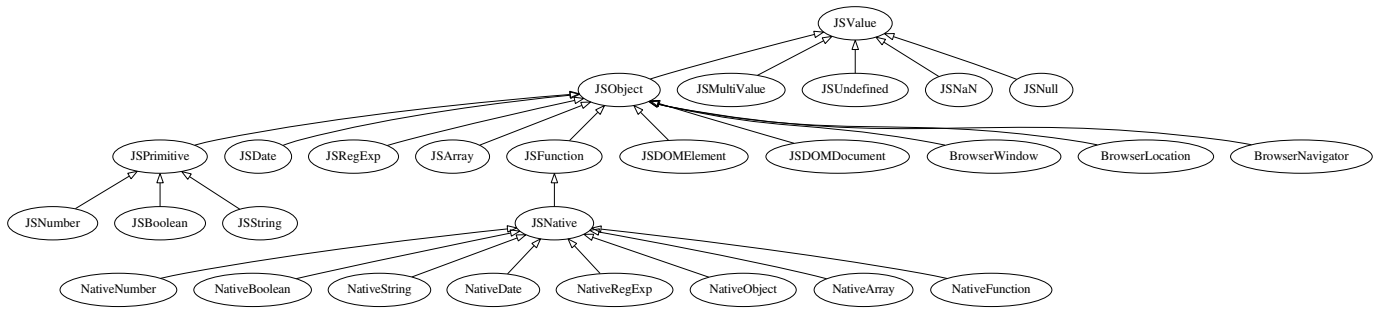


Fig. 5: Java class inheritance hierarchy representing the modeled JavaScript types.

codes to be analyzed and saves half of the network time to download them (Section VI-F).

Take jQuery as an example, which is by far the most popular library, appearing in more than half of all websites that use JavaScript [2]. The following gives the syntax of some jQuery APIs that indicate the occurrence of URLs or dynamically injected HTMLs. We have identified a total number of nine rules for jQuery.

```
jQuery.ajax(url [, settings])
jQuery.get(url [,data] [,success(data, status, jqXHR)])
$('#DOM_ELEM_ID').attr("src", url)
$('#DOM_ELEM_ID').attr("href", url)
$('#DOM_ELEM_ID').html(html)
```

The matching rules for the above jQuery APIs are shown below.

```
jQuery.ajax(url={0})
jQuery.get(url={0})
$(*).attr({0}="src", url={1})
$(*).attr({0}="href", url={1})
$(*).html(html={0})
```

In the matching rule syntax, `url` and `html` are keywords indicating the occurrence of a URL or a piece of HTML code. `{0}` represents the value of the first argument and so forth. `{0}="src"` defines a constraint that the value of the first argument must equal to the string value "src". For example, the third matching rule means that the second argument passed to `attr` is a URL if the first argument equals to "src". During the execution, if this rule is satisfied, the value of the second argument is recorded as a URL.

Supporting other JavaScript libraries, similarly, is to define matching rules for each library and store them in a configuration file.

E. Retrieving Web Page Contents

To fetch page contents of a given URL, we build a resource downloader, which creates an HTTP or HTTPS connection to a URL resource and then reads all the contents of the remote object if successfully connected.

HTTP Redirects. A browser makes a redirection when it receives a certain HTTP status code that responds to a Get, Post or Head type of web request. For example, if we make a request to "http://wikipedia.org", we will receive an HTTP 301

response, indicating the page is moved permanently to "http://www.wikipedia.org". (see Fig. 6 for the HTTP response header) from the remote server, with the HTTP status code being 301 — meaning "moved permanently". The destination URL of redirection is provided by the `Location` header entry of the HTTP response.

HTTP/1.1 301 moved permanently	
Location	http://www.wikipedia.org/
Content-Length	233
Content-Type	text/html; charset=iso-8859-1
Date	Mon, 26 Mar 2012 13:41:44 GMT
Server	Apache

Fig. 6: The HTTP response header of making a request to "http://wikipedia.org".

Our resource downloader is configured to automatically follow HTTP redirects and record the final landing URL, which will be used to build the absolute URL for each relative URL discovered.

The Rejected Request Problem. Websites like google.com reject connection requests that are not initiated by a browser to prevent potential malicious accesses. We call this phenomenon the rejected request problem. It is necessary to keep our resource downloader from being disturbed by this problem. To this end, the resource downloader mimics a real browser by inserting several headers when sending requests, as shown in Fig. 7.

Tiding up malformed HTML. We use `jsoup`⁶, a Java library for working with real-world HTML, to tidy up malformed HTML.

⁶<http://jsoup.org/>

User-agent	Mozilla/5.0 (Windows NT 6.1; rv:6.0) Gecko/20100101 Firefox/6.00
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language	en-us,en;q=0.5
Accept-Encoding	gzip, deflate
Accept-Charset	ISO-8859-1,utf-8;q=0.7,*;q=0.7

Fig. 7: The HTTP request header mimicking the Firefox browser.

F. Slicing

A great deal of client-side JavaScript code is concerned with presentation details. In finding URLs hidden in client-side JavaScript programs, it is a good idea to concentrate our effort on program portions where URLs are bound to occur. If we could safely determine, using syntactic criteria, which code is irrelevant to our analysis, we could skip a large volume of it. This would give us the scalability we will need as client-side JavaScript programs grow in size and complexity. Based on this observation, we perform slicing on JavaScript programs in order to remove URL-irrelevant code and produce an executable program slice that is able to produce equal number of URLs as the original program.

Slicing Criteria. Informally, program slicing [14] is to compute the set of program statements—the program slice, that may affect the values at some point of interest, referred to as a slicing criterion. In the context of this work, a slicing criterion is a JavaScript statement that contains a variable whose value is a URL string. Section III-B gives the taxonomy of URL-relevant program points in a client-side JavaScript program. According to this categorization, we traverse the AST and along the way identify seed statements for slicing.

Slicing Algorithm Configuration. IBM’s WALA [15] provides a series of static analysis capability for programming languages including JavaScript. WALA includes a slicer that operates on the intermediate representation (IR) defined by WALA itself. We use this slicer to compute a backward slice. The configuration for pointer analysis is needed before slicing. The *ZeroOneCFA* policy is chosen, which provides an approximation of Andersen’s pointer analysis [16], using allocation sites to name abstract objects. For data dependency options, we choose `NO_HEAP_NO_EXCEPTION`. As to control dependency options, `NO_EXCEPTIONAL_EDGES` is set to ignore exceptional control flow edges.

Dealing with the Result of Slicing. The output of WALA’s slicing is a set of WALA IR instructions, which is rather low-level. However, our execution interpreter is based on Rhino AST nodes⁷. The slicing result is not useful unless it is converted to an AST of Rhino nodes. We consider translating the slicing result to Rhino AST through the source program, since both WALA and Rhino provide source code mapping facilities — WALA provides the mapping between an IR instruction and its corresponding line number in the original source program, while Rhino similarly provides the mapping between an AST node and its corresponding line number in the original source program. For an AST node, if there is not any IR instruction in the slice mapping to the line number referred to by this AST node, this AST node is removed; otherwise, it is kept. Therefore, in order to make full use of the slicing result, the source program (the input of slicing) should be formatted to avoid the situation that many IR instructions map to the same source line number. This is achieved by passing

⁷Readers might wonder why not to further implement the simulator with WALA. This is because WALA is designed for program analysis tasks, but is neither expected to nor supports any form of code generation or execution, as stated by WALA developers.

the source program that is formatted by calling the `toSource` method on the AST root (mentioned in Section V-B).

VI. EVALUATION

We compare our approaches described in Section IV against a purely dynamic approach as well as a script-agnostic approach. For the hybrid approach, we further divide it into the *Hybrid* scheme described in Section IV-C, the hybrid with statement coverage (*Hybrid+SC*), the hybrid with range analysis (*Hybrid+RA*), and the one with both techniques (*Hybrid+SC+RA*).

For the *script-agnostic* approach, we use the open source Weblech [17] spider, which simply ignores JavaScript in HTML documents and only extracts URLs from HTML tags. This is the way most web crawlers do.

For the purely dynamic approach (*browser*), we use the SWT browser [18] of the Eclipse IDE. The SWT browser allows a client to turn on or off its JavaScript engine before loading a web page. In the experiment, we use the SWT browser to load a web page once with JavaScript disabled and a second time with JavaScript enabled. URLs that only exist in the latter are considered as dynamically generated by JavaScript.

All our experiments were conducted on a machine with an Intel Core 2 Duo 2.93 GHz CPU, 4 GB memory.

A. Datasets

Our dataset for testing consists of two sets of web pages: a list of top sites and a list of hot pages provided by alexa.com on July 13, 2012. The rank of top sites is calculated using a combination of average daily visitors and page views over the past month [19]. Hot pages are the most popular pages based on the data collected by the Alexa toolbar. These two sets allow us to obtain representative web pages for our analysis on JavaScript.

Specifically, the top sites dataset (called *Top* hereinafter) consists of home pages of 2,000 top global sites. The hot pages dataset (called *Hot* hereinafter) are 2,000 popular pages not included in the list of top sites. Table II gives a summary of the two datasets. We can observe that both have similar sizes of HTML documents. Hot pages have a significantly larger size of statically included JavaScript and reference more external JavaScript files than top sites.

TABLE II: Summary of the two datasets. HTML and JavaScript sizes are in KB.

	HTML Size	JS Size	# of External JS
Top	70	233	9
Hot	73	370	14

B. Metrics

We use three metrics to evaluate the performance of different approaches, i.e., the number of URLs extracted, accuracy, and running time.

The extracted URLs consist of static URLs (extracted from the HTML document) and dynamic URLs (discovered by analyzing the JavaScript). These two kinds of URLs are stored in separate sets and the total URLs are URLs union the two sets. Therefore, a URL is only counted once even if it occurs multiple times.

Accuracy is defined as the proportion of true positives against all positive results. In this paper, false positives are URLs found by an approach but judged as invalid in a URL validation process.

URL validation is in two steps. First, a regular expression is used to test if a string is in a valid URL format. If the string passes this test, then we try to retrieve the URL content. If the HTTP response code is 200 (which means OK) and the content length is greater than zero, the URL is considered as valid. Otherwise, the URL is considered as a false positive.

The running time is defined as the interval from the time an input URL is given to the time analysis is done, including the time to fetch page contents from remote servers.

C. Overall Performance

Table III reports the overall experimental results of all approaches. The columns from the left to right are: the total number of URLs extracted, average number of URLs extracted per page, the total number of valid URLs, average number of valid URLs per page, the total accuracy of all URLs, the accuracy of dynamic URLs, total running time, average running time per page, average running time to extract a URL and the average running time for a valid URL. The hybrid approaches get the largest number of valid URLs, followed by the AST-based approaches, the text-based approach and the browser-based approach. As expected, the script-agnostic crawler finds the least number of valid URLs.

The hybrid approach with statement coverage and range analysis finds 39.4% more valid URLs than the script-agnostic approach. We note that statement coverage is more effective than range analysis, achieving 10.0% (versus 1.6%) improvement over the hybrid scheme. When both techniques are used, the improvement is 15.8% over the hybrid scheme. The AST-compute approach is better than AST-literal approach, indicating there are a number of URLs that are constructed with only a few steps of computation without context information.

In terms of accuracy of extracted JavaScript URLs, the browser approach is more accurate than all other approaches, because the usage of a real browser engine. The text-based approach is the least accurate, because simple pattern matching introduces many false positives.

For running time, the browser approach is the slowest one and the script-agnostic browser performs the best. The fact that the browser approach spent more than half a second to extract one valid URL on average demonstrates that such a dynamic approach is too expensive to use. All other approaches perform similarly, about twice the time of the script-agnostic approach, due to downloading external JavaScript files. In fact, the network downloads account for 90% of the total running time

on average, while less than 10% time is spent on extracting URLs from a web page.

D. Effectiveness of Modeling HTML DOM

To understand the contribution of modeling HTML DOM, we compare the performance of the hybrid approaches with and without DOM support.

TABLE IV: Performance of the hybrid approach with and without DOM support.

	without DOM		with DOM	
	Total	Valid	Total	Valid
Hybrid	1019153	934621	1026724	941419
Hybrid+RA	1029479	942316	1046725	953809
Hybrid+SC	1109991	1004420	1127687	1017642
Hybrid+SC+RA	1154564	1040598	1190532	1062769

Table IV shows that when DOM is enabled, all approaches perform better than their counterparts without DOM support. In fact, modeling DOM helps the *Hybrid+SC+RA* approach to extract 2.9% more valid URLs. This experiment indicates that DOM plays an important role in JavaScript code for constructing URLs.

E. Study on HTML Inline Scripts

This experiment studies the effectiveness of only analyzing HTML inline scripts, where no external JavaScript files are downloaded.

TABLE V: Performance with only inline scripts analyzed.

	Total	Total valid	Total time(ms)	Per valid URL(ms)
Text-based	67165	48007	30014455	37.33
AST-literal	59573	50085	29984924	37.20
AST-compute	70889	60042	29994004	36.76
Hybrid	86626	71986	30128673	36.38
Hybrid+RA	84359	78112	30256869	36.27
Hybrid+SC	107399	86895	30508132	36.20
Hybrid+SC+RA	131033	105755	31920335	37.04

Table V shows that just analyzing the inline script is effective, which yields 6.6% to 13.9% more URLs than the script-agnostic approach. Because no external JavaScript files are downloaded in this experiment, the total running time is comparable to the script-agnostic approach.

Because downloading external JavaScript finds 39.4% more URLs, it is clear that external JavaScript files provide more useful information. However, including those files will significantly slow down the analysis process, due to high networking cost.

F. Impact of Analyzing JavaScript Libraries

In this paper, we used a pattern matching approach during the execution to handle JavaScript libraries, which saves time for both downloading these files from network and analyzing the library code. This experiment studies the impact of analyzing these libraries.

Table VI illustrates the results of analyzing all library code. It can be observed that this approach can actually find more

TABLE III: Overall performance results on all web pages.

	Extracted URLs		Valid URLs		Accuracy		Running Time(ms)			
	Total	Per page	Total	Per page	Total	JS URL	Total	Per page	Per URL	Per Valid
Script-agnostic	807101	202	758820	190	94.02%	-	29952554	7488	37.11	39.47
Browser	920648	230	857926	214	93.19%	87.28%	490601142	122650	533.00	571.97
Text-based	983641	250	882493	221	89.72%	70.05%	71874218	17968	73.17	81.56
AST-literal	961193	240	887966	222	92.38%	84.09%	71956341	17989	74.97	81.15
AST-compute	988666	247	911511	228	92.20%	83.81%	72140258	18035	73.07	79.26
Hybrid	1026724	257	941419	235	91.69%	82.90%	75085686	18771	73.23	79.87
Hybrid+RA	1046725	261	953809	238	91.12%	81.37%	77167423	19292	73.82	81.01
Hybrid+SC	1127687	281	1017642	254	90.24%	80.73%	77683574	19420	68.97	76.44
Hybrid+SC+RA	1190532	297	1062769	265	89.27%	79.27%	88780272	22195	74.66	83.63

TABLE VI: Performance with all scripts analyzed.

	Total	Total valid	Total time(ms)	Per valid URL(ms)
Text-based	1012278	897014	114798988	128.10
AST-literal	983162	906014	118593014	131.01
AST-compute	1008526	927935	120895848	130.40
Hybrid	1041192	950412	141098224	148.00
Hybrid+RA	1064851	965655	157183473	162.88
Hybrid+SC	1154474	1038284	177988394	171.53

URLs. For instance, *Hybrid+SC* finds 2.7% more valid URLs. However, this increase comes with a cost of significantly longer execution time. Our pattern matching approach achieves a speedup ranging from 1.5 to 2.2. This is because our approach saves half of the network time to download external JavaScript files and avoids the effort to analyze more than 70% of the total JavaScript code which is actually irrelevant to our analysis.

G. Analysis on Dynamic URLs

Distribution of Static URLs. Figure 8 shows the distribution of static URLs in HTML documents. Here, `a:href` stands for URLs embedded in the `href` attribute of the `a` tag and so forth. `css:url`, however, represents URLs of resources defined in the contents of `style` tags.

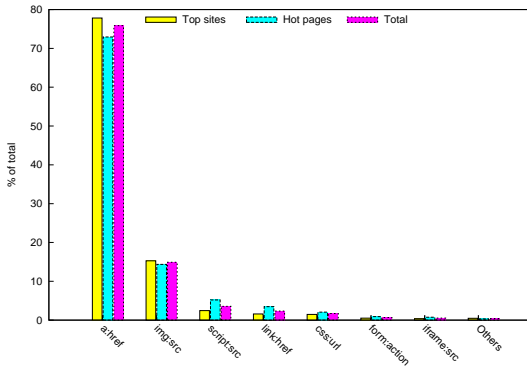


Fig. 8: Distribution of Static URLs

Distribution of Dynamic URLs. A large portion of the dynamically generated URLs is used to update or insert elements into the HTML document. We compared the HTML document loaded with JavaScript enabled and the HTML document loaded with JavaScript disabled by the SWT browser and counted the places of dynamic URLs in the HTML document.

Fig. 9 shows that more than 50% dynamic URLs are for anchors. About 15% of the dynamic URLs are used to dynamically include JavaScript by inserting a new `script` tag, which is a common practice for lazy loading to reduce user-perceived latency, but complicates program analysis. Another 16% are image URLs, apparently for lazy loading too. Finally, about 6% are used to render the HTML by inserting `style` tags.

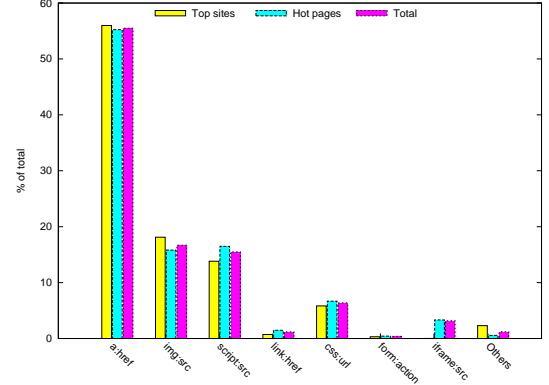


Fig. 9: Distribution of dynamic URLs.

Distribution of URL-relevant Points. Fig. 10 shows the distribution of URL-relevant points recorded by our JavaScript interpreter. We classify function calls and property assignments to the `window` object and `location` object into window-related and location-related, respectively. Property assignments to the `src` or `href` attributes are counted together.

Fig. 10 indicates that more than half of the dynamic URLs are generated by updating the DOM tree, such as updating the `href` attribute of an anchor element or inserting new elements contains URLs via `document.writeln`. Redirection accounts for slightly more than 3% of the URLs.

Study on jQuery API Usage. We also recorded function calls to jQuery APIs where URLs occur during the execution. Fig. 11 shows the distribution of the usage of such jQuery APIs. It can be observed that `$.get`, `$.ajax`, and `$.getScript` are the top three used methods, accounting for more than half of the calls accessing network resources.

H. Discussion on Accuracy

Though effective and efficient, the accuracy of our hybrid approaches is lower than that of the script-agnostic approach.

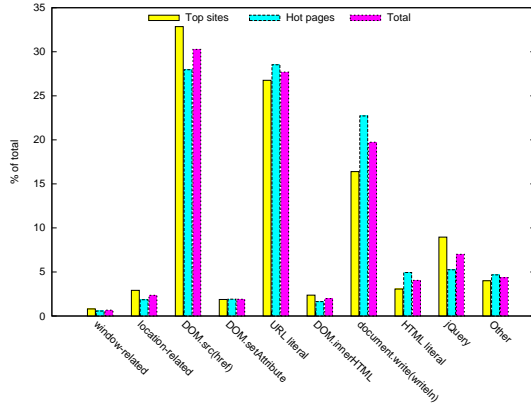


Fig. 10: Distribution of URL-relevant points.

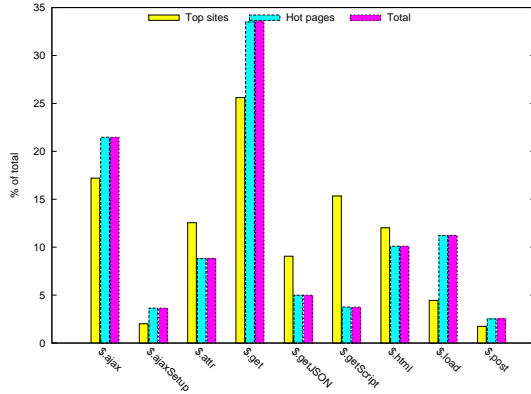


Fig. 11: Usage of jQuery APIs.

Except for imprecision in interpreted execution, there are other reasons for the reported false positives.

- There is a non-negligible period of time between downloading web pages and carrying out analysis on them. During this period of time, a web resource once existed might be removed, causing the corresponding URL to be judged as a false positive during validation. This is especially true for dynamically constructed URLs because they tend to be short-lived.
- To validate all URLs collected from a certain website, we send HTTP connection requests to the remote server. When the number of requests gets large, the IP sending the requests is likely to be blocked by remote sites in prevention of DoS attacks, causing the requests to fail.
- Our analysis produces the correct URL, but the corresponding web resource cannot be accessed. For instance, the web server encounters internal errors or returns 404 (not found).

For these reasons, quite a few among the false positives are actually valid URLs. Note that even the script-agnostic approach only has 94.02% accuracy, indicating our accuracy metric is a conservative estimation. The real number of valid URLs and accuracy should be higher than reported.

I. Threats to Validity

VII. RELATED WORK

We summarize related work as follows.

Detection of JavaScript Redirection Spam. Redirection spam detection also relies on finding URLs in web pages. Wu et al. [20] conducted a preliminary study on the distribution of clocking and redirection spam among: HTTP 301, HTTP 302, META refresh and navigation made by JavaScript. However, they only detect `location.replace` and `window.location` substrings in the script tags of the web page. As we have discussed, the use of text-based approach can easily introduce incorrect values. Chellapilla et al. [5] summarized techniques of JavaScript-based redirections and detected redirections by running a real browser. Their findings show that JavaScript obfuscation techniques are very prevalent in JavaScript redirection spam, which limit the effectiveness of static analysis and static feature based system. They recommended the use of a lightweight JavaScript parser and a tuned JavaScript execution environment for redirection spam detection, which inspires our work to implement a self-defined JavaScript interpreter. In our implementation, we also take into account the advanced redirection techniques they summarized, including script injection, HTML injection and event injection. Thomas et al. [21] designed a real-time system that crawls URLs as they are submitted to web services and determines whether the URLs direct to spam content. They employed an instrumented version of Firefox to collect features for URL classification. We show that this dynamic execution approach is very expensive and they may miss redirection information in unvisited path, because the browser can execute only one path of the program. Our approach can be used as a lightweight alternative to detect JavaScript redirections.

AJAX Crawling. Several previous work focuses on crawling AJAX contents. Mesbah el al. [22] proposed a dynamic approach to infer a state-flow graph model of the various navigation paths and states within an AJAX application. The model is used to generate a multi-page static version of the original AJAX application. The work of [23] presented a model of AJAX web sites and introduced an architecture of an AJAX search engine. They described the algorithms of AJAX crawling, indexing and query processing. Our work, to some extent, can also crawl AJAX. However, AJAX crawling focuses on indexing the dynamic AJAX content while ignoring the detail of AJAX communication and updates to the DOM. Our work, instead, focuses on finding all the URLs of AJAX communication. How the DOM tree is updated from the AJAX response and how to index and search the AJAX content are not our concerns.

Static Analysis of JavaScript. Early work on static analysis of JavaScript code has focused on the language itself and often for restricted subsets of the language. This body of work includes analysis on type systems [24], [25], [26] and points-to analysis [27]. Jensen et al.'s work [8] is closely related to our work. They model HTML DOM and browser API to develop a static analysis that is capable of reasoning about the

control flow and data flow in client-side JavaScript. Our work also models the HTML DOM and browser objects. However, our work doesn't model the event system. Meanwhile, static analysis techniques of JavaScript are developed and applied to attack problems in the security research domain. Saxena et al. [28] presented a symbolic-execution framework for exploring the execution space of JavaScript. They applied their tool to the problem of finding client-side code injection vulnerabilities. Guha et al. [1] used a control-flow analysis to extract a model of the client behavior in an AJAX application as seen from the server, and build an intrusion-prevention proxy for the server. This group of research differs from ours in that our defining goal is to spot variables that represent URLs and acquire their values, which requires sufficient precision in analysis, and thus can only be achieved by running the program.

Improvement on Execution Coverage. Cove et al. [29] combined anomaly detection with emulation to automatically identify malicious JavaScript code. At run time, they parse all the code provided to the JavaScript interpreter and keep track of all the functions that are defined therein. When the regular execution finishes, they force the execution of functions that have not been invoked. Our execution sequence, however, will first execute all identified event handlers then execute the functions that have not been invoked after the regular execution. Moser et al. [12] proposed a system that allows exploring multiple execution path and identify malicious actions that are executed only when certain conditions are met. The multiple execution paths technique, as they called, is actually the same with our statement coverage technique. They applied some optimization that we haven't implemented.

Empirical Study of JavaScript. There is also empirical research on the practical use of JavaScript. Yue and Wang [30] presented a study on insecure practices of using JavaScript on the web. Richards et al. conducted a large-scale study of the runtime behavior of JavaScript programs, whose results challenge quite a few commonly made assumptions about the dynamic behavior of JavaScript [31]. In their later work [2], they studied the use of `eval` and showed that the categories of `eval` are often used.

VIII. CONCLUSIONS

With the increase of dynamic content on the web, crawlers and search engines cannot afford to be script agnostic. Our analysis starts with a taxonomy of URL-embedding patterns in JavaScript. Then we present several approaches to analyze and extract embedded URLs in client-side JavaScript. In particular, we have presented a hybrid approach that first employs static analysis to reason about control flows, and then uses dynamic execution to overcome challenges caused by JavaScript language's dynamic features, especially the `eval`-like functions and interactions with HTML DOM.

Our experimental results demonstrate that the hybrid approach is highly productive in extracting embedded URLs from client-side JavaScript with a reasonable amount of running time. For top sites and popular pages, our tool

finds 39.4% more URLs in JavaScript code. We also see strong evidence that JavaScript is responsible for lazy loading page contents, calling for attentions from search engines. Experimental results show that statement coverage combined with range analysis can boost the performance of the hybrid approach by 15.8%.

The results from this research can not only be applied to web crawlers to improve their coverage, but also be used for other applications such as URL filtering or redirection spam detection. Our future work is to improve the precision and performance of our program analysis.

REFERENCES

- [1] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for Ajax intrusion detection," in *Proceedings of the 18th International Conference on World Wide Web (WWW)*. ACM, 2009, pp. 561–570.
- [2] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do," in *Proceedings of the 25th European conference on Object-oriented programming (ECOOP)*. Springer, 2011, pp. 52–78.
- [3] Google, "Cloaking, sneaky javascript redirects, and doorway pages," <http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=66355/>, 2011.
- [4] W3C, "HTML 4.01 Specification," <http://www.w3.org/TR/html401/>, 1999.
- [5] K. Chellapilla and A. Maykov, "A taxonomy of JavaScript redirection spam," in *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*. ACM, 2007, pp. 81–88.
- [6] J. Gruber, "An improved liberal, accurate regex pattern for matching URLs," http://daringfireball.net/2010/07/improved_regex_for_matching_urls/, 2010.
- [7] D. Flanagan, *JavaScript: The Definitive Guide, Sixth Edition*. O'Reilly Media, April 2011.
- [8] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the HTML DOM and browser API in static analysis of JavaScript web applications," in *Proceedings of the 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 59–69.
- [9] Google, "Chrome V8 introduction," <https://developers.google.com/v8/intro>, 2012.
- [10] Wikipedia, "Rhino (JavaScript engine)," [http://en.wikipedia.org/wiki/Rhino_\(JavaScript_engine\)](http://en.wikipedia.org/wiki/Rhino_(JavaScript_engine)), 2013.
- [11] —, "SpiderMonkey (JavaScript engine)," [http://en.wikipedia.org/wiki/SpiderMonkey_\(JavaScript_engine\)](http://en.wikipedia.org/wiki/SpiderMonkey_(JavaScript_engine)), 2013.
- [12] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP)*, 2007, pp. 231–245.
- [13] Google, "Closure Compiler," <http://code.google.com/p/closure-compiler/>, 2012.
- [14] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*. IEEE Press, 1981, pp. 439–449.
- [15] S. Fink and J. Dolby, "WALA-The T. J. Watson Libraries for Analysis," <http://wala.sourceforge.net/>, 2012.
- [16] L. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, University of Copenhagen, 1994.
- [17] "WebLech URL Spider," <http://weblech.sourceforge.net/>.
- [18] "SWT: The standard widget toolkit," <http://www.eclipse.org/swt/>.
- [19] Alexa Internet Inc., "Alexa-the Web Information Company," <http://www.alexa.com/>, 2012.
- [20] B. Wu and B. D. Davison, "Cloaking and redirection: A preliminary study," in *Proceedings of the First International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, 2005, pp. 7–16.
- [21] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song, "Design and evaluation of a real-time URL spam filtering service," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP)*, 2011, pp. 447–462.
- [22] A. Mesbah, E. Bozdag, and A. v. Deursen, "Crawling AJAX by inferring user interface state changes," in *Proceedings of the 2008 Eighth International Conference on Web Engineering (ICWE)*, 2008, pp. 122–134.

- [23] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou, “AJAX Crawl: Making AJAX applications searchable.”
- [24] C. Anderson, S. Drossopoulou, and P. Giannini, “Towards type inference for JavaScript,” in *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, July 2005, pp. 428–452.
- [25] P. Thiemann, “Towards a type system for analyzing JavaScript programs,” in *Proceedings of the 14th European Symposium on Programming (ESOP)*, 2005, pp. 408–422.
- [26] S. H. Jensen, A. Møller, and P. Thiemann, “Type analysis for JavaScript,” in *Proceedings of the 16th International Symposium on Static Analysis (SAS)*, 2009, pp. 238–255.
- [27] D. Jang and K. Choe, “Points-to analysis for JavaScript,” in *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*. ACM, 2009, pp. 1930–1937.
- [28] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for JavaScript,” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*, 2010, pp. 513–528.
- [29] M. Cova, C. Kruegel, and G. Vigna, “Detection and analysis of drive-by-download attacks and malicious JavaScript code,” in *Proceedings of the 19th international conference on World Wide Web (WWW)*, 2010, pp. 281–290.
- [30] C. Yue and H. Wang, “Characterizing insecure JavaScript practices on the web,” in *Proceedings of the 18th international conference on World Wide Web (WWW)*, 2009, pp. 961–970.
- [31] G. Richards, S. Lebesne, B. Burg, and J. Vitek, “An analysis of the dynamic behavior of JavaScript programs,” in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 2010, pp. 1–12.